**Flipped Session Short Answer – Week 2**

Instructions: Work with a group (2-6 people) to answer as many questions as possible.

1. Give a concise description of the fundamental difference between a linked list of nodes and an array.

2. In the resizing array implementation of the stack, how does the line s[N] = null help reduce memory usage?

3. Is it a good idea to include a peek() method? Peek lets you see what is next on the stack without actually removing it.

4. Arrays have functionality that Stacks do not utilize, in particular the ability to access items anywhere in the array in constant time (including the middle). Does this suggest that arrays are a poor choice since they aren't fully utilizing the power of the array? Why or why not?

5. Does every operation take constant time in the worst case for the fixed-array-size implementation of stacks?

6. Is there any advantage to downsizing the ResizingArrayStack array other than saving memory?

7. Is the memory savings gained by shrinking the ResizingArrayStack array worth the cost in time?

8. One could argue that when you downsize an array, you're probably going to have to upsize it later. Why bother downsizing?

9. What do you think about the idea of providing the user of your Stack class with the ability to manually specify when to increase/decrease the size of the array?

10. A double ended queue is more flexible than both the stack and the queue and would use almost the exact same implementation. Why don't we just make all queues double ended (capable of adding and removing items from either end)?

11. Is the placement of the beginning and end arbitrary? Is there an advantage to doing things as in the lecture notes over the alternate: adding nodes to the start of the list and removing them from the end?

12. When else might we use generics other than to implement stacks and queues?

13. We saw that we could iterate through our Stack with code like:

    **for (String s : stack)**
    **  StdOut.println(s);**

    Why would we ever use an Iterator<> instead of the code shown above? (I'm not sure I fully understood this question, but it was certainly popular on FlipYourTraining).

14. What uses are there for the Bag data structure? Wouldn't it be simpler to just use a stack or queue and never pop?

15. "Why is there Iterable and Iterator? Couldn't both be wrapped up in a single structure without losing any functionality, rather than having to retreive an extra class and use that?"

16. Our sorting algorithms take as input an array of Comparable[]. To be a Comparable, an object of type TypeX must simply implement the Comparable interface. One student asked ' couldn't you technically put anything into this array, even if it isn't comparable with the other elements, as long as it is from a class that implements Comparable?'. Why is this not an issue in Java?

17. Would we be able to make a class that is comparable to other types of objects? For example, could we make a class similar to String that could also be compared to integers? How would we do this?

18. "Does an interface do more than just list the functions that must be implemented? I understand that it's a very useful tool to have a consistent list of functions across objects, but would it ever cause an error to avoid implementing the interface if you would still implement all the necessary functions?"

19. For shellsort, what is bad about the increment sequence 1, 2, 4, 8, 16, 32, …?

20. Insertion performs .25N^2 comparisons and .25N^2 exchanges is .5N^2 operations, which seems comparable to selection sort's .5N^2 comparisions and N (lower order and thus unimportant) exchanges. Yet Bob mentioned that we expected selection sort to be twice as fast based on the comparisons. Did Bob misspeak?

21. Extra: Is it actually possible to implement precedence order with the Dijkstra two stack algorithm?